# Exploiting Thread-Level and Instruction-Level Parallelism for Hyper-Threading Technology

Xinmin Tian
Senior Staff Engineer
Software and Solutions Group
Intel Corporation

Aart Bik
Senior Staff Engineer
Software and Solutions Group
Intel Corporation

Milind Girkar
Principal Engineer
Software and Solutions Group
Intel Corporation

Paul Grey
Staff Software Engineer
Software and Solutions Group
Intel Corporation

# Table of Contents

(Click on page number to jump to sections)

## EXPLOITING THREAD-LEVEL AND INSTRUCTION-LEVEL PARALLELISM FOR HYPER-THREADING TECHNOLOGY

# Exploiting Thread-Level and Instruction-Level Parallelism for Hyper-Threading Technology

Xinmin Tian
Senior Staff Engineer
Software and Solutions Group
Intel Corporation

Aart Bik
Senior Staff Engineer
Software and Solutions Group
Intel Corporation

Milind Girkar
Principal Engineer
Software and Solutions Group
Intel Corporation

Paul Grey
Staff Software Engineer
Software and Solutions Group
Intel Corporation

## *Overview*

Perhaps the most compelling reason for exploring Hyper-Threading Technology is its ability to accommodate a high utilization of processor resources. A Pentium® 4 processor that is enabled with Hyper-Threading Technology, for example, enables the exploitation of parallelism at both thread-level and instruction-level.

Thread-level parallelism is exploited with multithreaded code that takes advantage of multiple control, arithmetic, and logical units. Instruction-level parallelism, on the other hand, is exploited with instructions of SSE and SSE2 (Streaming-SIMD-Extensions) by vectorizing loops that apply a single operation to multiple elements in a data set. This article provides a high-level overview of the parallelization and vectorization technology used by the Intel® C++/Fortran compiler developed at the Intel Compiler Labs.

## *Compiler Architecture*

A high-level overview of the Intel C++/Fortran compiler architecture is shown in **Figure 1**. The compiler incorporates many well-known and advanced optimization techniques that fully leverage features of Intel processors for higher performance. The compiler has a common intermediate representation (called IL0) for the programming languages C++, C and Fortran95. As a result, OpenMP* directive-guided or pragma-guided parallelization, automatic loop parallelization and vectorization, and the majority of other optimizations are applicable through a single, high-level code transformation that is independent of the source language. Throughout the rest of this article, Intel C++ and Fortran compilers for the IA-32 and Itanium® processor family architectures will be collectively referred to as “the Intel compiler.”

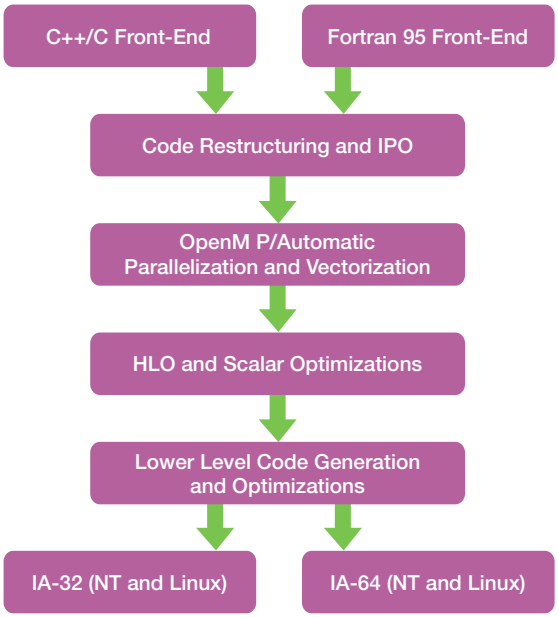


**Figure 1. Compiler Architecture Overview**

The code transformations and optimizations in the Intel compiler can be classified into:

- Code restructuring and interprocedural optimizations (IPO).
- OpenMP directive/pragma-guided/automatic parallelization and automatic vectorization (PAROPT).
- High-level optimizations (HLO) and scalar optimizations, including loop control and data transformations, partial redundancy elimination (PRE), and partial dead store elimination (PDSE).
- Low-level machine code generation and optimizations such as register allocation and instruction scheduling.

Parallelization guided by OpenMP directives, pragmas, or detected automatically by the compiler with data dependency and control-flow analysis is a high-level code transformation that exploits both medium- and coarse-grained parallelism on multiprocessors and uniprocessors that are enabled with Hyper-Threading Technology to achieve better performance and higher throughput. The intermediate representation IL0 has been extended to express the OpenMP directives and pragmas. Implementing the OpenMP phase at the IL0 level allows the same implementation to be used across languages (C++/C, Fortran95) and architectures (IA-32 and Itanium). The generated code references a high-level multithreaded library application programming interface (API). This allows the compiler multithreaded code generation phase to be independent of the underlying operating systems and facilitates a “one-for-all” design philosophy.

## *Thread-Level Parallelism*

Exploiting thread-level parallelism (TLP) is an effective way to improve the performance of applications with the advent of general-purpose, cost-effective uniprocessor and multiprocessor systems. The following two subsections give an overview of the multithreading technology in the Intel compiler.

*OpenMP Pragma-Guided Multithreading*

The Intel compiler supports OpenMP directive- and pragma-guided parallelization, which significantly increases the domain of various applications amenable to effective parallelism. In a typical example, programmers use OpenMP parallel sections to develop an application where *section-1* calls an integer-intensive routine and *section-2* calls a floating-point intensive routine. In this manner, performance improvement is obtained by scheduling *section-1* and *section-2* onto two different logical processors that share the same physical processor to fully utilize processor resources based on the Hyper-Threading Technology.

The OpenMP 2.0 standard API supports a multiplatform, shared-memory, parallel programming paradigm in C++/C and Fortran95 on all Intel architectures and popular operating systems such as Windows NT*, Linux*, and UNIX*. The OpenMP directives/pragmas have emerged as the de facto standard of expressing thread-level parallelism in various applications, as they substantially simplify the notoriously complex task of writing multithreaded applications. Below is a sample program using the *parallel sections* pragma.

*(1) A Program Example with Parallel Sections*

```
void  parfoo( )
{
   int  m, y, x[5000];
   float   w, z[3000];

#pragma omp parallel sections shared(w, z, y, x)
  {
      w =  floatpoint_foo(z, 3000);
    #pragma omp section
       y =  myinteger_goo(x, 5000) ;
  }
}
```

Essentially, the multithreaded code generator inserts the thread invocation call *__kmpc_fork_call*(…) with *T-entry* node and data environment (source line information *loc*, thread number *tid*, etc.) for each parallel loop, parallel sections or parallel region, and transforms a serial loop, sections, or region to a multithreaded loop, sections, or region, respectively. In this example, the *pre-pass* first converts *parallel sections* to a *parallel loop*. Then, the multithreaded code generator localizes loop lower-bound and upper-bound, and privatizes the section *id* variable for the T-region marked with [T_entry, T-ret] nodes.

For the *parallel sections* in the routine "*parfoo*", the multithreaded code generation involves (a) generating a runtime dispatch and initialization routine (*__kmpc_dispatch_init*) call to pass necessary information to the runtime system; (b) generating an enclosing loop to dispatch *loop-chunk* at runtime through the *__kmpc_dispatch_next* routine in the library; and (c) localizing the loop lower-bound, upper-bound, and privatizing the loop control variable '*id* ' as shown below.

*(2) Pseudo-Threaded-Code After Parallelization*

```
R-entry void  parfoo( )
{ … …
  __kmpc_fork_call(loc, 4, T-entry(__parfoo_psection_0), &w, z, x, &y)
  goto L1:
  T-entry void __parfoo_psection_0(loc, tid, *w, z[], *y, x[]) {
    lower = 0;   upper = 1; stride = 1;
    __kmpc_dispatch_init(…, tid, lower, upper, stride, …);
   L33:
    t3 = __kmpc_dispatch_next(..,, tid, &lower, &upper, &stride)
    if ((t3 & upper>=lower) != 0(SI32)) {
           pid = lower;
       L17:  if (pid == 0) {
                  *w = floatpoint_foo(z, 3000);
             } else if (pid == 1) {
                 *y = myinteger_goo(x, 5000);
             }
             pid = pid +1;
             __kmpc_dispatch_fini(…);
             if (upper >= pid)  goto L17
       goto L33
    }
    T-return;
   }
L1:  R-return;
}
```

Because the granularity of the sections could be dramatically different, the static or static-even scheduling type may not achieve good load balancing. We decided to use the *runtime* scheduling type for a parallel loop generated by the pre-pass of multithreaded code generation. Therefore, the decision regarding scheduling type is deferred until run-time, and an optimal balanced workload can be achieved based on the setting of the OMP_SCHEDULE.

The Intel compiler also supports the *taskqueuing* model that allows the user to parallelize control structures that are beyond the scope of those supported by the OpenMP model, while attempting to fit into the framework defined by

OpenMP. In particular, the *taskqueuing* model is a flexible mechanism for specifying units of work that are not precomputed at the start of the worksharing construct. For OpenMP *single*, *for* and *sections* constructs, all work units that can be executed are known at the time the construct begins execution. The *taskqueuing* pragmas *taskq* and *task* relax this restriction by specifying an environment (the taskq) and the units of work (the tasks) separately.

The *taskq* pragma specifies the environment within which the enclosed units of work (tasks) are to be executed. From among all the threads that encounter a *taskq* pragma, one is chosen to execute it initially. Conceptually, the *taskq* pragma causes an empty queue to be created by the chosen thread, and then the code inside the *taskq* block is executed single-threaded. All the other threads wait for work to be enqueued on the conceptual queue. The *task* pragma specifies a unit of work, potentially executed by a different thread. When a *task* pragma is encountered lexically within a *taskq* block, the code inside the task block is conceptually enqueued on the queue associated with the *taskq.* The conceptual queue is disbanded when all work enqueued on it finishes, and when the end of the *taskq* block is reached. Many control structures exhibit the pattern of separated work iteration and work creation, and are naturally parallelized with the taskqueuing model. Some common cases are C++ iterators, while loops, and recursive functions.

*Automatic Loop Multithreading*

Automatic threading is a promising technique for taking advantage of an Intel Hyper-Threading Technology–enabled Pentium 4 processor or Intel Xeon™ processor-based multiprocessor that can deliver good performance gains. However, threading inner loops usually does not provide sufficient granularity of parallelism. Therefore, our focus for automatic threading is to exploit medium-grained parallelism to utilize a Hyper-Threading Technology–enabled processor or multiprocessor system effectively. Finding effective parallelism is one of the critical steps in generating efficient multithreaded code. The Intel compiler takes the following steps:

- Find all loops within the serial code and build a loop hierarchy structure. Fill up loop parameters such as trip count, lower bound, upper bound, and pre-header.
- Perform data dependence analysis to classify loops. Loops without loop-carried dependency are marked as parallelizable loops.
- Perform static or dynamic granularity estimation for each parallelizable loop. Multithreaded code for a parallelizable loop will be generated if and only if this loop is a profitable parallel loop.
-

*An example of this optimization is shown below.*

```
for (k=0; k < 1000; k++) {
     x[k] = k;
     w    = x[k];
     y[k] = w + x[k];
}
```

*Auto loop-threading detection marks this loop as follows.*

```
parallel for (k=0; k < 1000; k++) {
     private (k, w), shared (x, y)

     x[k] = k;
     w    = x[k];
     y[k] = w + x[k];
}
```

In this example, the loop is marked as a parallelizable loop for generating threaded code, and the variables ‘k’ and ‘w’ are marked as **private**. The arrays ‘x’ and ‘y’ are marked as **shared** by the compiler. Parallelizing a loop can result in slower execution if the overhead of dispatching/scheduling threads and sharing resources is significant compared to the total workload performed by the loop. The Intel compiler handles this by examining all the operations in the loop body, estimating the grain-size per loop iteration on the targeted microarchitecture, and multiplying this by the loop trip count to arrive at an estimate of the total workload of the loop.

For loops with known trip counts, this value is compared at compile-time to an experimentally determined profitable workload threshold, to see if the loop should be multithreaded. Loops with a workload exceeding this profitable threshold will normally speed up when executed in parallel threads. For loops with unknown trip counts, the workload is expressed as a function of the trip count and the compiler generates code to dynamically evaluate this expression to

determine whether the loop should be executed with multiple threads. Note that this solution avoids all dispatching/scheduling overhead and sharing resources if multithreaded execution is not profitable.

### *Instruction-Level Parallelism*

Recent multimedia extensions to the Intel architecture provide an alternative way to utilize data parallelism in multimedia and scientific applications. These extensions let multiple functional units operate simultaneously on packed data elements, i.e., relatively short vectors that reside in memory or registers. The Pentium 4 processor features SSE and SSE2 (Streaming-SIMD-Extensions) that support floating-point operations on four packed single-precision and twp packed double-precision floating-point numbers, as well as integer operations on 16 packed bytes, eight packed words and four packed dwords. The Intel compiler supports the automatic conversion of sequential code into SIMD form, a transformation that here is referred to as intra-register vectorization.

Combining intra-register vectorization with parallelization for hyper- or multithreading enables the exploitation of multilevel parallelism, i.e., using different forms of parallelism that are available in an application to obtain high performance. Take, for instance, the following code for matrix-vector multiplication.

*(1) Thread-Level and Instruction-Level Parallelism Example:*

```
double a[N][N], x[N], y[N];
…
#pragma omp parallel for private(k,j)
 for (k = 0; k < N; k++) { /* parallel loop */
   double d = 0.0;
   for (j = 0; j < N; j++) { /* vector loop */
       d += a[k][j] * y[j];
   }
   x[k] = d;
 }
…
```

*(2) Pseudo code after Parallelization and Vectorization*

```
   __kmpc_fork_call(loc, 0, T-entry(_ompvec_ploop_0),  … )
   goto L1:
   T-entry _ompvec_ploop_0(loc, tid) {
     lower = 0;
     upper = N;
     __kmpc_static_init(loc, tid, STATIC, &lower, &upper, ...);
     prv_k = lower;
L2:
     xorpd   xmm0, xmm0             ; reset accumulator
L3:
     movapd   xmm1, _a[ecx+edx]     ; load 2 DP  from a
     mulpd    xmm1, _y[edx]         ; mult 2 DP  from y
     addpd    xmm0, xmm1            ; add  2 DP  into accumulator
     add      edx, 16               ;
     cmp      edx, eax              ;
     jl             L3              ; looping logic

     movapd    xmm1, xmm0           ;
     unpckhpd  xmm1, xmm1           ;
     addsd     xmm0, xmm1           ; compute final sum

     store result in  x[prv_k]

     prv_k = prv_k + 1
     if (prv_k <= upper) goto L2;

     __kmpc_static_fini(loc, tid);
     T-ret;
  }
L1:  … …
```

In this example, parallelism appears at multiple levels. The iterations of the outermost *k-loop* may execute independently, as has been made explicit with an OpenMP pragma. The reduction performed in the innermost *j-loop* provides yet another level of parallelism. This loop can be implemented by accumulating partial sums in SIMD style, followed by code that constructs the final sum. In (E6-II), you can see how these two levels of parallelism can be exploited (where it is assumed that all access patterns in the vector loop are aligned at a 16-byte boundary).

If the alignment of memory references cannot be determined at compile-time, the Intel compiler has at its disposal several alignment optimizations to avoid performance penalties that are usually associated with unaligned memory access. Dynamic data dependence testing is used to allow the compiler to proceed with vectorization in situations where analysis has failed to prove independence statically. These advanced techniques (and others) have been discussed in detail in previous work.

## Performance

In this section, we show some performance results for the matrix-vector multiplication kernel discussed above on a Hyper-Threading Technology–enabled Intel Xeon processor dual-CPU system running at 1.5 GHz with 512MB of memory, an 8K L1-cache and a 256K L2-cache. **Figure 2** shows speed-ups (relative to serial execution) for varying matrix sizes for vector execution (VEC), multithreaded execution using two threads and four threads, (OMP2) and (OMP4), respectively, and vector-multithreaded execution using two and four threads, (OMP2+VEC) and (OMP4+VEC), respectively.

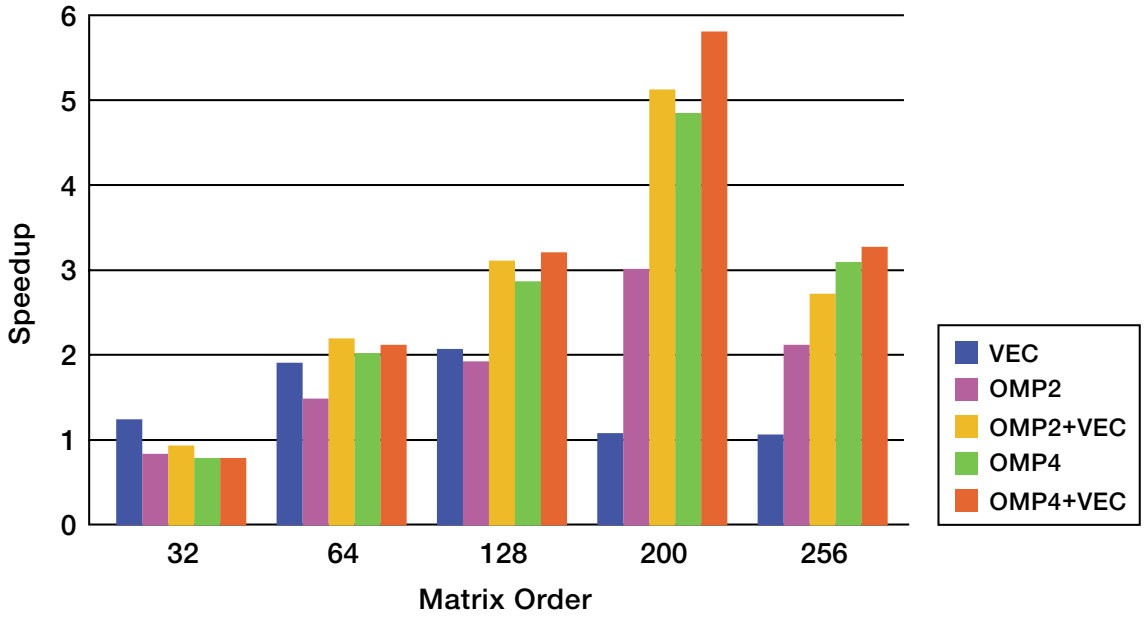


Figure 2. Performance of Matrix X Vector Kernal

Timings were obtained by calling the kernel many times and dividing the total execution time accordingly, which implies that for the data sets that completely fit in cache, the kernel is computationally bound. In these cases, intra-register vectorization alone obtains a speed-up of up to 2x. For the larger data sets, where the kernel becomes more memory bound, the improvements of intra-register vectorization become less evident. The overhead associated with multithreading causes a slight slowdown for the matrix size 32x32. For the larger matrices ranging from 64x64 to 256x256, the relative overhead introduced by parallelization becomes negligible, and observed speed-up ranges from 1.4x to 5.8x.

The difference between (OMP2) and (OPM4) for matrix size 200x200 reveals a 1.6x performance gain. For the same matrix size, the performance gain from the versions that are optimized with intra-register vectorization, (OMP2+VEC) and (OMP4+VEC), is 1.2x. The best performance gains are obtained when all levels of parallelism (SIMD parallelism and parallelism due to Hyper-Threading Technology and multithreading) are exploited simultaneously, yielding a speed-up of up to 5.8x with four threads (OMP4+VEC) and a speed-up of 5.1x with two threads (OMP2+VEC).

## Summary

In this article, we have provided a high-level overview of the automatic parallelization and vectorization technology used by the Intel® C++/Fortran compiler developed at the Intel Compiler Labs. We have attempted to show that these methods can obtain good speed-up on a Hyper-Threading Technology–enabled Pentium 4 processor with minimum effort. Hence, Intel's compiler technology provides a convenient way for programmers who are not familiar with the Intel architecture to boost the performance of their applications.

In addition, it may assist expert programmers by minimizing the number of loops or regions that have to be hand-optimized to exploit all available parallelism. Finally, the approach allows for the parallelization and vectorization of existing serial software, thereby avoiding potentially large investments that would be required to optimize this software.

## More Info

More information on Intel's high-performance compilers can be found at the Intel® Software Development Products Web site and at the OpenMP Web site. Information for software developers about Intel's Hyper-Threading Technology can be found at the Hyper-Threading Technology Web site. The February 2002 issue of the Intel Technology Journal also featured more information about this and other Hyper-Threading-related topics.

## Author Bios

Xinmin Tian is a senior staff engineer in the vectorization and parallelization group of the Intel Compiler Labs, part of the Software and Solutions Group. He is currently working on compiler parallelization and optimization, and manages the OpenMP Parallelization group. He holds B.S., M.S., and Ph.D. degrees in computer science from Tsinghua University. He was a postdoctoral researcher in the School of Computer Science at McGill University, Montreal. Before joining Intel, he worked on parallelizing compiler, code generation, and performance optimization at IBM.

Aart Bik is a senior staff engineer in the Intel Compiler Labs, part of the Software and Solutions Group. He received his M.S. in computer science from Utrecht University, The Netherlands, and his Ph.D. degree from Leiden University, The Netherlands. In 1997, he was a postdoctoral researcher at Indiana University, where he conducted research in high-performance compilers for Java*. In 1998, he joined Intel, where he is currently working in the vectorization and parallelization group.

Milind Girkar is the principal engineer in the IA-32 Compiler Development group of the Intel Compiler Labs, part of the Software and Solutions Group. He received a B.Tech. degree from the Indian Institute of Technology, Mumbai, an M.S. from Vanderbilt University, and a Ph.D. in computer science from the University of Illinois at Urbana-Champaign. Currently, he manages the IA-32 Compiler Development group. Before joining Intel Corp., he worked on an optimizing compiler for the UltraSPARC* platform at Sun Microsystems.

Paul Grey is a staff software engineer in the Intel Compiler Labs, part of the Software and Solutions Group. He received his B.S. degree in applied physics at the University of the West Indies, and his M.S. in computer engineering at the University of Southern California. Currently he is working on compiler optimizations for parallel computing. Before joining Intel, he worked on parallel compilers, parallel programming tools, and graphics system software at Kuck and Associates, Inc., Sun Microsystems, and Silicon Graphics, Inc. He is interested in optimizing compilers, advanced microarchitecture, and parallel computers.

*—End of Intel Developer Update Magazine Article—*